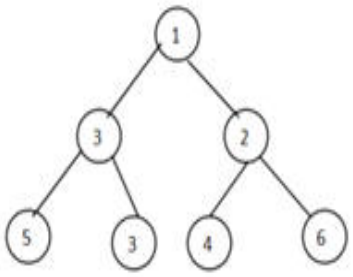
**Abstract Data Types - Bags (Multisets)**

A bag, or multiset, is a collection of elements where the order is immaterial but, unlike a set, duplicate elements do matter. For example, [1,2,2,3] does not equal [1,2,3] but [1,2,2,3] does equal [2,1,2,3].

We define the operations on bags to include:

| Function | Explanation |
|---|---|
| `makeBag :: [a] -> Bag a` | convert a list to a bag. |
| `isEmpty :: Bag a -> Bool` | determines if a bag is empty |
| `union :: Bag a -> Bag a -> Bag a` | union of two bags |
| `minBag :: Bag a ->  a` | returns the minimum value in the bag |
| `deleteMin :: Bag a -> Bag a` | removes one occurrence of the min value in the bag |

We could implement bags using a sorted lists but a more efficient representation uses a data structure called a *heap*. A heap is a binary tree such that the root is the smallest value in the tree and the two branches are also heaps. We will extend this a bit to make sure that the number of nodes in the left heap is at least as large as the number of nodes in the right heap. The following is an example of a heap.




The following code implements the Bag ADT with a heap.

```
module Bag (Bag, isEmpty, minBag, union, deleteMin, makeBag) where

data Heap a = Null | Branch Int a (Heap a) (Heap a) deriving Show
type Bag a = Heap a

isEmpty :: Bag a -> Bool
isEmpty Null = True
isEmpty (Branch n x left right) = False

minBag :: Bag a -> a
minBag (Branch n x left right) = x
```

```
union :: (Ord a) => Bag a -> Bag a -> Bag a
union Null y = y
union x Null = x
union (Branch m u l1 r1) (Branch n x l2 r2) =
     if u <= x then
          branch u l1 (union r1 (Branch n x l2 r2))
     else
          branch x l2 (union (Branch m u l1 r1) r2)

-- branch takes an element and two subtrees and creates a Heap with the
-- additional property that the left subtree is at least as big as the right
-- subtree
branch :: a -> Bag a -> Bag a -> Bag a
branch x left right =
     if (size left) < (size right) then
          Branch newSize x right left
     else
          Branch newSize x left right
where newSize = (size left) + (size right) + 1

size :: Bag a -> Int
size Null = 0
size (Branch n x left right) = n

deleteMin :: (Ord a) => Bag a -> Bag a
deleteMin (Branch n x left right) = union left right

makeBag :: (Ord a) => [a] -> Bag a
makeBag xs = fst(makeTwo (length xs) xs)

makeTwo :: (Ord a) => Int -> [a] -> (Heap a, [a])
makeTwo 0 xs = (Null,xs)
makeTwo 1 (x:xs) = (branch x Null Null, xs)
makeTwo n xs = (union x y, zs) where
    (x, ys) = makeTwo m xs
    (y, zs) = makeTwo (n- m) ys
    m = n `div` 2
```

What is the efficiency for each of the operations?